OlympusDAO Security Review

Auditors

Christoph Michel, Lead Security Researcher

Desmond Ho, Lead Security Researcher

Blackscale, Security Researcher

Jonatas Martins, Apprentice

Hagrid, Apprentice

Report prepared by: Pablo Misirov and Jonatas Martins

July 24, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

The goal of OlympusDAO's Bophades protocol is to create a decentralized, censorship-resistant reserve currency for the emerging Web3 ecosystem. Developing a reserve currency is important because a fundamental goal of the Web3 financial movement is to foster an alternative economic ecosystem that serves the needs of its various stakeholders. This follow-up specialized review by Spearbit focussed on the alternative economic ecosystem, governance, and voting system.

The Bophades protocol is designed with Model-View-Controller (MVC) framework and it is architecturally separated into Module and Policy structures, where Modules contain data and Policies contain business logic.

The focus of this security review included the following, but was not limited to:

- Detecting general architecture vulnerabilities.
- Examining *markets* are opened and closed properly.
- Focusing on how *module* and *policy* upgrades affect the system.
- Checking if any miscalculations on market operations exist.
- Checking if the protocol synchronizes correctly during state changes.
- Controlling that if all modules have valid and strict permissions.
- Abusing the protocol to benefit from any loss of funds.
- Validating the *RANGE* system works correctly.
- Optimizing gas usage to increase performance.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of OlympusDAO Bophades2 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, OlympusDAO engaged with Spearbit to review Bophades2. In this period of time a total of 59 issues were found.

Summary

Project Name	OlympusDAO
Repository	Bophades2
Commit	87b3252e7af93d034e8bf ...
Type of Project	DeFi, Framework
Audit Timeline	Jun 22nd - July 4th
Methods	Manual Review, Dynamic Analysis

Issues Found

Critical Risk	4
High Risk	6
Medium Risk	10
Low Risk	12
Gas Optimizations	12
Informational	15
Total Issues	59

5 Findings

5.1 Critical Risk

5.1.1 Wrong capacity on bond market creation

Severity: *Critical Risk*

Context: Operator.sol#L318, Operator.sol#L386

Description: The capacities are specified in OHM on the high side and Reserve on the low side. However, the bond market creation in `_activate` uses the wrong capacities:

- high side: `capacityInQuote: true` and quote token is `reserve`
- low side: `capacityInQuote: true` and quote token is `ohm`

Recommendation: Consider changing `capacityInQuote` to be `false` for both sides.

Olympus: Fixed Operator.sol#L337 for the _activate function.

Spearbit: Acknowledged.

5.1.2 Incorrectly implemented revert invalidates adding or upgrading modules

Severity: *Critical Risk*

Context: INSTR.sol#L98-104

Description: It is not possible to store new modules or upgrade an existing module on the `INSTR.sol` contract due to an invalid revert condition. In addition, the `Governance.sol` contract makes an external call to `INSTR.store()` via the `submitProposal()` function. As a result, Governance cannot add nor upgrade modules since execution will always revert.

Recommendation: Remove the unnecessary revert.

```
if (
  instruction.action == Actions.InstallModule ||
  instruction.action == Actions.UpgradeModule
) {
    Module module = Module(instruction.target);
    _ensureValidKeycode(module.KEYCODE());
-   revert INSTR_InvalidChangeExecutorAction();
}
```

Olympus: Fixed, INSTR.sol#L100

Spearbit: Acknowledged.

5.1.3 Wrong capacity updates when swapping on high side

Severity: *Critical Risk*

Context: Operator.sol#L255

Description: The capacity is specified in OHM on the high side. However, when swapping on the high side (`tokenIn_ == reserve`) the function reduces capacity by `amountIn` minus a reserve token amount. Capacity updates on the high side are wrong leading to an incorrect OHM amount which can be bought from the high wall. The wall on the high side does not work as expected.

Recommendation: Reduce the high wall capacity by the OHM amount.

```
} else if (tokenIn_ == reserve) {
  /// Revert if lower wall is inactive
  if (!RANGE.active(true)) revert Operator_WallDown();

  /// Calculate amount out (checks for sufficient capacity)
  amountOut = getAmountOut(tokenIn_, amountIn_);

  /// Decrement wall capacity
- _updateCapacity(true, amountIn_);
+ _updateCapacity(true, amountOut);

  /// If wall is down after swap, deactive the cushion as well
  _checkCushion(true);

  /// Transfer reserves to treasury
  reserve.safeTransferFrom(msg.sender, address(TRSRY), amountIn_);

  /// Mint OHM to sender
  MINTR.mintOhm(msg.sender, amountOut);
}
```

Olympus: I had made a (probably too hasty) change late in the process to switch the high side capacity to OHM units instead of reserve units. This is a relic from that, along with the `fullCapacity` issue. Fixed in Operator.sol#L272

Spearbit: Acknowledged.

5.1.4 Callback is vulnerable to a Denial Of Service attack by sending arbitrary amount of quote tokens

Severity: *Critical Risk*

Context: BondCallback.sol#L95-L99

Description: The callback can be easily exploited with a DOS attack by directly sending any amount of `quoteToken` to the contract because it checks for strict equality between the expected and actual amount received.

Recommendation: Consider introducing the following change:

```
if (
-  quoteToken.balanceOf(address(this)) !=
+  quoteToken.balanceOf(address(this)) <
   priorBalances[quoteToken] + inputAmount_
) revert Callback_TokensNotReceived();
```

Olympus: Good catch. We'll update the check. Fixed in BondCallback.sol#L112

Spearbit: Acknowledged.

5.2 High Risk

5.2.1 Incorrect check allows anyone to revoke any policy's token approval

Severity: *High Risk*

Context: TRSRY.sol#L135-L136

Description: The function is callable by anyone to revoke a policy's token approval. However, the check that the policy is terminated is performed on `msg.sender` instead of `withdrawer_`. This therefore allows anyone to revoke any policy's token approval, blocking any policy from operating properly with the treasury.

Recommendation: The intention might have been to check `withdrawer_` instead of `msg.sender`.

```
- if (kernel.approvedPolicies(msg.sender) == true)
+ if (kernel.approvedPolicies(withdrawer_))
    revert TRSRY_PolicyStillActive();
```

However, this still doesn't fit the function's description: "Anyone can call to revoke a terminated policy's approvals". Checking if a policy is *terminated* can be done by checking if it exists in `allPolicies` and `approvedPolicies[policy]` should be false. However, this check is quite inefficient as it requires iterating over `allPolicies`.

Consider reworking this function and making it restricted to the `APPROVER`.

Olympus: We will rethink this function. We do not like the coupling of policy needs directly to the treasury. We will likely make something like a Treasury Manager policy that includes this function, and make it an `APPROVER` role.

Fixed. Also created a TreasuryCustodian policy that has this function instead, as this should not be core function of treasury - TreasuryCustodian.sol

Spearbit: This still allows anyone to remove approvals on non-policy spenders (instead of only on terminated policies).

Olympus: You are correct. I split the function out but forgot to address the issue completely. I will put a comment and leave as acknowledged for now, since we're changing how the policies and modules are stored inside the kernel due to some of the findings from the audit, which will be out of scope.

Spearbit: Acknowledged, fix has not been implemented.

5.2.2 Capacity does not account for token precision

Severity: *High Risk*

Context: Operator.sol#L701-L705

Description: The `fullCapacity` function should account for reserve and ohm decimals precision. Consider removing the `mulDiv` function as well to improve efficiency as `FACTOR_SCALE` would need to be unreasonably large (but is currently fixed at `1000`), or the price would need to go to extremes that would make other parts of the (eco)system fail before the contract hits this limit.

Recommendation: Consider implementing the following change.

```
- capacity = capacity
-     .mulDiv(10**PRICE.decimals(), RANGE.price(true, true))
-     .mulDiv(FACTOR_SCALE + RANGE.spread(true) * 2, FACTOR_SCALE);
+ capacity =
+   capacity *
+     10**ohm.decimals() *
+     10**PRICE.decimals() /
+     10**reserve.decimals() /
+     RANGE.price(true, true) *
+     (FACTOR_SCALE + RANGE.spread(true) * 2) /
+     FACTOR_SCALE;
```

Olympus: Fixed, Operator.sol#L779

Spearbit: Acknowledged.

5.2.3 Can buy more than capacity & cause a Denial Of Service due delayed market capacity updates

Severity: *High Risk*

Context: Operator.sol#L560

Description: The `capacity` includes the allowance for the bond market (`marketCapacity`) but it's only lazily-updated with the used market capacity whenever calling the `_updateCapacity` function:

```
function _updateCapacity(bool high_, uint256 reduceBy_) internal {
    /// Get the market for the side
    uint256 market = RANGE.market(high_);

    /// Initialize update variables, decrement capacity if a reduceBy amount is provided
    uint256 capacity = RANGE.capacity(high_) - reduceBy_;
    uint256 marketCapacity;

    /// If the market is active, adjust capacity and market capacity
    if (auctioneer.isLive(market)) {
        /// Get current market capacity
        marketCapacity = auctioneer.currentCapacity(market);

        /// Reduce capacity by the amount of capacity the market has expended sicne the last update
+       // @audit lazy update, capacity could already be close to 0 because of `swap` and underflow
        capacity -= RANGE.lastMarketCapacity(high_) - marketCapacity;
    }

    /// Update capacities on the range module for the wall and market
    RANGE.updateCapacity(high_, capacity, marketCapacity);
}
```

Meaning, users can buy more than the `capacity` & break the contract.

Example: Assume `capacity = 1000`, `marketCapacity = cushionFactor(20%) * 1000 = 200`, threshold capacity ("dust" value that capacity needs to drop below to set range to inactive) `threshold = capacity * thresholdFactor(1%) = 10`. Assume there is a bond market and price is close to wall price.

User calls `swap` and uses 990 capacity. Range is not inactive as `newCapacity = 1000 - 990 = 10 < 10 = threshold` is `false`. As range is not inactive, market is not deactivated in `_checkCushion`. Therefore, the user can still buy the entire 200 market capacity from the bond market (`auctioneer.purchaseBond`). User used up `990 + 200 = 1190` capacity in total. When the next `operate -> _updateCapacity` heart beat is executed this subtraction underflows (`10 - (200 - 0) = -190`), the transaction reverts, the contract is temporarily broken.

Recommendation: Consider addressing this by:

1. Any market capacity used by the market must update the `RANGE` capacity immediately, instead of lazy updating here.
2. Have separate capacities for swap and bonds that don't interfere with each other. (Right now `marketCapacity` is part of `capacity`.)

Olympus: We could have the `BondCallback` automatically update the capacity on each purchase. It may be easier to just change the capacity check in swap to take out the deployed market capacity. I'll discuss with the team.

Olympus: Operator.sol#L628 for _updateCapacity function in Operator.sol

RANGE.sol#L131 for updateCapacity function in RANGE

Spearbit: The behavior is a bit different now. The market is already closed as soon as the entire capacity dropped (due to swaps) below the market capacity. But this seems to be the intention according to the comment. Fixed.

5.2.4 Incorrect comparison leads to wrong decimals derivation

Severity: *High Risk*

Context: Operator.sol#L423-L427 **Description:** The conditional check performed to derive the number of decimals in the price is `while (price_ > 10)`, which means that prices starting with `10...` will have 1 decimal less.

For instance, if `price = 1e18`, the calculated `decimals` is `17`.

Recommendation: Consider applying the following change.

```
- while (price_ > 10)
+ while (price_ >= 10)
```

Olympus: That makes sense. We'll update the check. Fixed Operator.sol#L486

Spearbit: Acknowledged.

5.2.5 Range updates are not applied immediately in `operate`

Severity: *High Risk*

Context: Operator.sol#L149-L176

Description: The `operate` function works on a cached `range` memory variable to determine whether to close or open bond markets. The `range` might not be up to date anymore as `_regenerate` calls might have happened in between. Bond markets are not immediately open or closed with the wall.

Recommendation: The protocol uses a policy and module contract structure with the module acting as the storage for one or more policies. It is important to always fetch the latest module data after a module call. Fetch the latest `range` again before determining whether markets should be activated or deactivated.

```
OlympusRange.Range memory range = RANGE.range();
Status memory status_ = _status;
Config memory config_ = _config;

/// Check if walls can regenerate capacity
if (
    uint48(block.timestamp) >=
    range.high.lastActive + uint48(config_.regenWait) &&
    status_.high.count >= config_.regenThreshold
) {
    _regenerate(true);
}
if (
    uint48(block.timestamp) >=
    range.low.lastActive + uint48(config_.regenWait) &&
    status_.low.count >= config_.regenThreshold
) {
    _regenerate(false);
}

/// Get latest price
/// See note in addObservation() for more details
uint256 currentPrice = PRICE.getLastPrice();

+ /// fetch the latest range again as it might have been updated in _regenerate
+ range = RANGE.range();

/// Check if the cushion bond markets are active
/// if so, determine if it should stay open or close
/// if not, check if a new one should be opened
if (range.low.active)
```

Olympus: Good catch. Yes, this is an issue. The range state should be updated after potential regenerations and before determining whether markets should be activated or deactivated. Fixed in L194-L195

Spearbit: Acknowledged.

5.2.6 Denial of Service on Governance actions

Severity: *High Risk*

Context: Governance.sol#L150

Description: The `endorseProposal` function does *not* lock user's voting tokens. A user can endorse a proposal, send the tokens to another account they control and vote again. It's easy for anyone to push a proposal above the endorsement threshold. Additionally, this function is not flashloan resistant.

There can only be one single active proposal at a time and an attacker can perform a DOS by:

- Creating a proposal.
- Endorsing it, sending tokens to another account under their control and repeating until the endorsement threshold is reached.
- Setting it as the active proposal which blocks any other proposals to become active for a week.
- Repeat this process so a proposal can never pass.

Recommendation: It's not obvious how to fix the DOS in the current code. You would have to lock tokens on endorsement too but then users will not be able to vote with these tokens on the active proposal anymore. Or you allow several active proposals.

Olympus: Yes, had the same thoughts. I was originally hesitant about locking tokens for endorsements because it makes the quorums much more volatile and difficult to reason about. I'm also not a fan of the active proposals because I think people can get too distracted/chaotic and that clarity and focus should be essential for on-chain governance because it's so inflexible.

One idea I've been playing with is paying a fee for the time that a proposal, kinda like the PoW model.

Spearbit: Some projects implement a slashing mechanism. The proposer stakes tokens together with proposal. If the proposal fails to garner sufficient votes then tokens will be burnt / distributed.

Olympus: We've resolved this issue by adding transfer locks to the voting token so they cannot be transferred. In addition, when the voting token goes live and we move to a staked token model, we will also implement warmup + cooldown periods to prevent any flash loan/similar type attacks to manipulate votes in the system.

Spearbit: Acknowledged.

5.3 Medium Risk

5.3.1 Markets are not always closed when range is deactivated

Severity: *Medium Risk*

Context: Operator.sol, RANGE.sol#L264

Description: There are certain edge cases where an open bond market is not closed when the `Range` is deactivated or regenerated. This happens whenever the `Range`'s `market` is set to a new value (`type(uint256).max` or a new market ID) without closing the previous market.

Consider the following proofs of concept:

Regenerating active wall does not close previous markets

It can be the case that an active range is close to regenerating (one more observation required to reach the threshold), a price move creates a bond market in the current `operate` call, the price in the next `operate` call changes and leads to the wall regenerating. The regeneration clears the bond market from the `range` struct but it is not closed through the `auctioneer`.

```
function test_marketFailToClose() public {
    /// Assert high wall is up
    assertTrue(range.active(true));

    /// Set price below the moving average to almost regenerate high wall
    price.setLastPrice(99 * 1e18);

    /// Trigger the operator function enough times to almost regenerate the high wall
    for (uint256 i = 0; i < 4; i++) {
        vm.prank(guardian);
        operator.operate();
    }

    /// Ensure market not live yet
    uint256 currentMarket = range.market(true);
    assertEq(type(uint256).max, currentMarket);

    /// Cause price to spike to trigger high cushion
    uint256 cushionPrice = range.price(false, true);
    price.setLastPrice(cushionPrice + 500);
    vm.prank(guardian);
    operator.operate();

    /// Check market is live
    currentMarket = range.market(true);
    assertTrue(type(uint256).max != currentMarket);
    assertTrue(auctioneer.isLive(currentMarket));

    /// Cause price to go back down to moving average
    /// Move time forward past the regen period to trigger high wall regeneration
    vm.warp(block.timestamp + 1 hours);

    /// Will trigger regeneration of high wall
    /// Will set the operator market on high side to type(uint256).max
    /// However, the prior market will still be live when it's supposed to be deactivated
    price.setLastPrice(95 * 1e18);
    vm.prank(guardian);
    operator.operate();
    /// Get latest market
    uint256 newMarket = range.market(true);

    /// Check market has been updated to non existent market
    assertTrue(type(uint256).max == newMarket);
```

```
    /// However, the current market is still live
    assertTrue(auctioneer.isLive(currentMarket));
}
```

Markets not closed after `swap` / `checkCushion` not working correctly

When `capacity` falls below the threshold through `swap` or auctioneer bond purchases the `updateCapacity` call will deactivate the `Range` and set the `range.market = type(uint256).max` without closing the previous market. The subsequent `_checkCushion` call will now check if the *new* market value (`type(uint256).max`) is still live instead of the old market value.

```
function test_marketFailToClose2() public {
    /// Assert high wall is up
    assertTrue(range.active(true));

    /// Set price on mock oracle into the high cushion
    price.setLastPrice(111 * 1e18);

    /// Trigger the operate function manually
    vm.prank(guardian);
    operator.operate();

    /// Get current market
    uint256 currentMarket = range.market(true);

    /// Check market has been updated and is live
    assertTrue(type(uint256).max != currentMarket);
    assertTrue(auctioneer.isLive(currentMarket));

    /// Take down wall
    knockDownWall(true);

    /// Get latest market
    uint256 newMarket = range.market(true);

    /// Check market has been updated to non existent market
    assertTrue(type(uint256).max == newMarket);
    /// However, the current market is still live
    assertTrue(auctioneer.isLive(currentMarket));
}
```

Recommendation: These faulty edge cases arise because it's hard for the `Operator` to know if a range was deactivated (and with it a market) in an `_updateCapacity -> RANGE.updateCapacity` call. The market is also overwritten whenever regenerating a range through `_regenerate -> RANGE.regenerate`. When these calls are performed, it must be checked if a new market was set and the old market must be closed in this case. (The old market could be cached before doing the call or `RANGE.updateCapacity/RANGE.regenerate` could return the old market.) Add test cases for the POCs to verify the fixes.

Olympus: This was fixed in conjunction with the updates to address in issue 39.

See Fixed this and applied the test cases (with the opposite assertion so they pass when fixed). Operator.sol#L723 and RANGE.sol#L198

Spearbit: Fix details Condition for deactivation was changed in _checkCushion() market set to _range.high/low.market instead of resetting to type(uint256).max. market capacity also no longer reset

Spearbit:

> `market` set to `_range.high/low.market` instead of resetting to `type(uint256).max`. market capacity also no longer reset

Same for `updateCapacity`. I personally find keeping the old code as comments there a bit confusing as it doesn't say *why* it's commented.

> These faulty edge cases arise because it's hard for the Operator to know if a range was deactivated (and with it a market) in an _updateCapacity -> RANGE.updateCapacity call.

The fix essentially addresses this by having the `RANGE` not change the market anymore in `regenerate/updateCapacity`.

For the first test case, when calling regenerate the market is now kept alive. However, the market's capacity is not reset but I think the correct behavior would be to do that?

Olympus: My intention was that the state transitions in the operate function make it where a market should not be active following a _regenerate call. More specifically, if the low side cushion is open, price moves up below the avg to trigger a regen, and operate is triggered, then the logic will close the previously active market anyways since the price is out of the lower cushion range. Same logic applies on the high side. With that in mind, the only potential issues are when a manual regenerate is triggered. Based on that, I believe I need to add a call to _deactivate in that function as well.

What do you think?

Spearbit: We believe issues come from the edge case that an already active range can be regenerated. Either through explicitly calling `regenerate` or by a sequence of events in `operate`. (the regenerate can be delayed because of the `regenWait` *time* threshold so it's in theory possible that there's 1) an active range, 2) price moves in other direction and collects active observations, 3) price moves back within cushion range and creates bond market, 4) next operate call triggers the regenerate because now `regenWait` finally passed. regen capacity is updated but not bond market's.)

In these cases, it could be that there's an active market that should be refreshed as if you called `_activate`?

> With that in mind, the only potential issues are when a manual regenerate is triggered. Based on that, I believe I need to add a call to _deactivate in that function as well.

Makes sense to let the next `operate` call reopen it if the price is within cushion range.

Olympus: Yeah, that's a good point. I think having _regenerate call _deactivate each time makes sense then. If the market is supposed to be opened, operate will re-open with an appropriate new capacity.

Olympus: Operator.sol#L713

Spearbit: Acknowledged.

5.3.2 Vote tokens locked for failed active proposal till another is made active

Severity: *Medium Risk*

Context: Governance.sol#L302-L305

Description: If the activate proposal cannot be executed (reverts due to say, time-sensitive operations or improper access control), voters are unable to reclaim votes until a new proposal takes over. This means that votes can potentially be locked for a long time should no subsequent proposal be made.

Recommendation: Add a voting window on the active proposal. Reclaiming after the voting window (even if it's not replaced with a new/blank proposal) should be fine then.

Olympus: We are going to resolve this issue by proposing a "blank" proposal that can be voted on to claim to allow locked tokens in a previously active proposal to be claimed.

Spearbit: Acknowledged.

5.3.3 Should use `safeTransfer` instead of `transfer`

Severity: *Medium Risk*

Context: BondCallback.sol#L136

Description: Using `transfer` function needs to check the return value, to fix this should use `safeTransfer`

Recommendation: Consider applying the following change.

```
- token.transfer(address(TRSRY), balance);
+ token.safeTransfer(address(TRSRY), balance);
```

Olympus: Fixed all occurences in BondCallback.sol.

Spearbit: Acknowledged.

5.3.4 Use of deprecated `latestAnswer()` function

Severity: *Medium Risk*

Context: PRICE.sol#L156 PRICE.sol#L159

Description: The `_ohmEthPriceFeed` is AggregatorV2V3Interface and it uses a deprecated `latestAnswer()` function. This method will return the last value but it is not possible to check if data is fresh according to Chainlink docs. It is also known that calling the `latestRoundData` allows to run some extra checks.

Recommendation: Use `latestRoundData()` function instead of deprecated `latestAnswer()`.

```
function latestRoundData() external view
    returns (
        uint80 roundId,
        int256 answer,
        uint256 startedAt,
        uint256 updatedAt,
        uint80 answeredInRound
    )
```

Olympus: Agree that this is an issue. We'll change to using the`latestRoundData()` function and check if the answer is older than a certain amount of time. Fixed in PRICE.sol#L161-L173.

Spearbit: Acknowledged.

5.3.5 Setting new `Regen` parameters breaks contract

Severity: *Medium Risk*

Context: Operator.sol#L508

Description: Setting a new `_config.regenObserve` value requires rescaling `_status.low/high.observations` to the new observe size. Otherwise, `operate -> _addObservation` might do an out-of-bounds write (if new observe > old observe) and thus breaking the contract.

Recommendation: Scale the `observations` arrays to the correct size initializing it with `false` values. Copy over the old values to the new array and set the `nextObservation` pointer correctly.

Olympus: Fixed in Operator.sol#L575-L581.

Spearbit: Acknowledged.

5.3.6 Users can end up `swap`ping at unexpected wall prices

Severity: *Medium Risk*

Context: Operator.sol#L223

Description: The `swap` function trades at the current wall price. This price is defined as a spread on the current moving average price. A `swap` can be (accidentally) frontrun by an `operate` call that updates the moving average and with it the wall price. The user's trade is executed at a different price and the user might receive fewer tokens than expected. It might be that the user would not have done the trade at the new price.

Recommendation: Add a `minAmountOut` parameter to the `swap` function and verify that the actual received amount `amountOut` is not less than `minAmountOut`.

```
function swap(ERC20 tokenIn_, uint256 amountIn_, uint256 minAmountOut_) {
  // ...
  require(amountOut >= minAmountOut, "slippage");
}
```

Olympus: That makes sense. I hadn't thought about the operate call front-running a trade since it happens rarely. We will add the slippage check. Fixed in Operator.sol#L254.

Spearbit: Acknowledged.

5.3.7 Callback is not protected from re-entrancy

Severity: *Medium Risk*

Context: BondCallback.sol#L117-L122

Description: `priorBalances` and `_amountsPerMarket` are updated after token transfers are performed, breaking the contracts-effects-interactions (CEI) pattern. It might therefore be possible to re-enter the callback if the payout token is ERC777 through the teller.

Recommendation: Add re-entrancy protection to the callback function.

Olympus: Fixed, BondCallback.sol#L98

Spearbit: Acknowledged.

5.3.8 Reclaiming votes for non-existent / future proposals not blocked

Severity: *Medium Risk*

Context: Governance.sol#L298-L300

Description: Voters can self-rug by reclaiming votes for non existent / future proposals. Since this sets their claims to true their tokens become permanently locked when these proposals become active and they vote on them.

Recommendation: Revert if `userVotes == 0`.

Olympus: Fixed in Governance.sol#L309-L310.

Spearbit: Acknowledged.

5.3.9 VOTES module upgrade will affect reclaiming of votes

Severity: *Medium Risk*

Context: Governance.sol#L315-L316

Description: Should the `VOTES` module be upgraded, there will always be a shortfall because the old `VOTES` token cannot be withdrawn. Voters who voted with the old `VOTES` token will receive the new one instead when reclaiming (assuming votes using the new `VOTES` token have been made).

Recommendation: If possible, the new `VOTES` token should initialize the Governance's balance as that of the existing `VOTES` token.

Olympus: This issue is more an ugprade/migration issue than a dependency issue, so we will resolve this issue in the migrations/VOTES module upgrades by minting the amount of new VOTE tokens to the Governance contract at the time of installation/upgrade to make up for the token shortfall.

Spearbit: Acknowledged.

5.3.10 Policy termination doesn't remove it from `allPolicies` array

Severity: *Medium Risk*

Context: Kernel.sol#L209

Description: `_terminatePolicy()` doesn't remove the policy from `allPolicies`. Thus, `allPolicies` will be non-decreasing and only increases over time. This has 2 consequences:

1) `_reconfigurePolicies()` might run out of gas as it iterates through the `allPolicies` array
2) A re-enabled policy will have its entry duplicated in the `allPolicies` array.

Recommendation: Remove the policy from `allPolicies` should it be terminated.

Olympus: The kernel is being rewritten currently to allow for this, but will also be out of scope for the remediation period.

Spearbit: Acknowledged.

5.4 Low Risk

5.4.1 TRSRY `setDebt` function must be used with with caution

Severity: *Low Risk*

Context: TRSRY.sol#L203-L217

Description: The function `setDebt` can be used to just debit anyone and needs to be used with caution. For example, one could set an arbitrary debt amount and it would increase the treasury as `getReserveBalance` includes debt.

Recommendation: A mentioned usecase for this function is to swap one reserve token (DAI) to another (WETH) and deposit the result into a yield farm. This could also be solved using functions that don't manually set debt:

- `withdrawReserves(trader, DAI, amount)`
- let trader contract do the swap to WETH
- send WETH to the treasury
- `loanReserves(WETH, amount)` and deposit it into the yield farm

Think about other usecases and consider if `setDebt` is necessary.

Olympus: Thank you for this. Agree it has to be a monitored function, but it is necessary. I think the case of swapping treasury assets is better with the recommendation you mentioned. But this function is still needed in

order to account for edge cases, like a potential debtor being hacked or being unable to return the funds (which happened to us with our funds deposited into Fuse last month).

Spearbit: Acknowledged.

5.4.2 `Price` setters requires new initialization

Severity: *Low Risk*

Context: File.sol#L123

Description: The `changeObservationFrequency` and `changeMovingAverageDuration` functions set the contract's `initialized` value back to `false`. This breaks all ongoing moving average updates and even the view functions like `getLastPrice()` and will make the entire system unresponsive until a new `initialize` call.

Recommendation: Immediately follow up with another `initialize` call by the guardian, ideally as part of the same transaction.

Olympus: This is intended behavior since the observations will be erased in many cases, but agree that it must be closely coordinated with a follow-up initialize.

Spearbit: Acknowledged.

5.4.3 `Price.observationFrequency` can get out of sync with `Heart.frequency`

Severity: *Low Risk*

Context: PRICE.sol#L49

Description: The code currently uses two frequencies to determine when the moving average price should be updated. The "real" frequency is the `Heart.frequency` which controls how often `beat()` and thus `updateMovingAverage` can be called by a keeper. The `Price.observationFrequency` is only used to compute the number of moving average observations `numObservations`. These two values can get ouf of sync when calling `changeObservationFrequency`. A desync leads to a wrong moving average computation as the number of observations changed but the actual frequency (`Heart.frequency`) with which the updates occur is still the same.

Recommendation: Consider removing `Heart.frequency` and let the `Price` module's `observationFrequency` be the only source of truth for frequency. A policy should not duplicate data that is already in a module.

Olympus: Fixed. Implemented in Heart.sol

Spearbit: Acknowledged.

5.4.4 Not updating `lastObservationTime` in PRICE

Severity: *Low Risk*

Context: PRICE.sol#L142

Description: `lastObservationTime` is commented and should be updated for accuracy. Future policies and external dependencies might rely on it.

Recommendation: Consider applying the following change.

```
- // lastObservationTime = currentTime;
+ lastObservationTime = currentTime;
```

Olympus: Fixed. Implemented in PRICE.sol#L143.

Spearbit: Acknowledged.

5.4.5 `setRewardToken` and `setReward` should be done at the same time

Severity: *Low Risk*

Context: Heart.sol#L135-L143

Description: Setting reward token and amount should be done at the same time. Otherwise, you might lose rewards. Imagine you're paying `100e18 DAI` rewards and trigger `setRewardToken(WETH) setReward(1e18)` but get frontrun on `setReward`. Keeper gets `100e18 WETH`

Recommendation: Consider applying the following change.

```
- function setReward(uint256 reward_) external requiresAuth {
-   reward = reward_;
- }

- /// @inheritdoc IHeart
- function setRewardToken(ERC20 token_) external requiresAuth {
-   rewardToken = token_;
-   emit RewardTokenUpdated(token_);
- }

+ function setRewardTokenAndAmount(ERC20 token_, uint256 reward_) external requiresAuth {
+   reward = reward_;
+   rewardToken = token_;
+   emit RewardTokenUpdated(token_);
+ }
```

Olympus: Fixed. Implemented in Heart.sol#L134-L141.

Spearbit: Acknowledged.

5.4.6 `beat` delays accumulate

Severity: *Low Risk*

Context: Heart.sol#L105-L108

Description: When calling `beat()` small delays accumulate (clock drift) and the moving average duration becomes inaccurate. For example, if it's called `10 seconds + frequency` after each `lastBeat`, the moving average accumulates `10 * numberOfObservations` delay.

Recommendation: Consider setting the `lastBeat` to the earliest possible call to counteract moving average clock drift.

```
/// Update the last beat timestamp
- lastBeat = block.timestamp;
+ lastBeat = lastBeat + frequency;
```

> Note: This change could lead to triggering several updates in a short time frame in the extreme case where the `beat()` call was missed for a long period of time.

Olympus: Implemented in Heart.sol#L101.

Spearbit: Acknowledged.

5.4.7 Heart contract can run out of reward tokens

Severity: *Low Risk*

Context: Heart.sol#L111

Description: The `beat()` function transfers reward tokens to the sender. If the contract has insufficient reward tokens then the `beat()` can't be called until it is refilled.

Recommendation: There are two approaches to this issue which depend on the reward token, in the case its an OHM token you can:

1. You can add `MINTER` rule to contract and mint new tokens
2. Store the rewards in a variable and the user can call a function later to get their rewards, this will unblock the beat function.

Olympus: Acknowledged. We anticipate this to require manual maintenance. We do not plan to use OHM for the keeper reward at this time so we cannot mint to reward them.

Spearbit: Acknowledged.

5.4.8 Ensure `observe_` is non-zero in setter function

Severity: *Low Risk*

Context: Operator.sol#L501-L503

Description: A check should be performed to ensure that `observe_ != 0`. Otherwise, doing `nextObservation = (regen.nextObservation + 1) % observe;` will revert because modulo by zero.

Recommendation: Consider applying the following change.

```
- if (wait_ < 1 hours || threshold_ > observe_)
+ if (wait_ < 1 hours || threshold_ > observe_ || observe_ == 0)
  revert Operator_InvalidParams();
```

Olympus: Fixed in Operator.sol#L566

Spearbit: Acknowledged.

5.4.9 Add checks to constructors

Severity: *Low Risk*

Context: Heart.sol#L61-L75 PRICE.sol#L68-L77 Operator.sol#L78-L88

Description: Variables in `Heart` and `PRICE` constructors can be invalid.

Recommendation: Consider adding additional checks.

In `Heart.sol`

```
constructor(
    Kernel kernel_,
    IOperator operator_,
    uint256 frequency_,
    ERC20 rewardToken_,
    uint256 reward_
) Policy(kernel_) Auth(address(kernel_), Authority(address(0))) {
+ if (frequency_ < 1 hours) revert Heart_InvalidParams();
    _operator = operator_;

    active = true;
    lastBeat = block.timestamp;
    frequency = frequency_;
    rewardToken = rewardToken_;
    reward = reward_;
}
```

In `PRICE.sol`

```
constructor(...) Module(kernel_) {
   /// @dev Moving Average Duration should be divislble by Observation Frequency to get a whole number
   ↪  of observations
-  if (movingAverageDuration_ % observationFrequency_ != 0)
+  if (movingAverageDuration_ == 0 || movingAverageDuration_ % observationFrequency_ != 0)
   revert Price_InvalidParams();
```

In `Operator.sol`

```
constructor(...) Policy(kernel_) Auth(address(kernel_), Authority(address(0))) {
+  if ( address(auctioneer_) == address(0) || address(callback_) == address(0)) revert
↪   Operator_InvalidParams();
+  if (configParams[1] > uint256(7 days) || configParams[1] < uint256(1 hours))
+    revert Operator_InvalidParams(); // TODO validate these bounds for duration
+  if (configParams[2] < uint32(10_000)) revert Operator_InvalidParams();
+  if (configParams[3] < uint32(1 hours) || configParams[3] > configParams[1])
+    revert Operator_InvalidParams();
+  if (configParams[4] > 10000 || configParams[4] < 100)
+    revert Operator_InvalidParams();
+  if (configParams[5] < 1 hours || configParams[6] > observe_)
+    revert Operator_InvalidParams();

...
}
```

Olympus: The Heart check was not applied because the frequency variable was removed from the Heart contract per other issue.

Spearbit: Acknowledged.

5.4.10 Swap amount rounding errors

Severity: *Low Risk*

Context: Operator.sol#L673-L686

Description: The `getAmountOut` function introduces rounding issues when converting the token amounts because it performs an early division.

```
uint256 amountOut = amountIn_
    .mulDiv(10**reserve.decimals(), 10**ohm.decimals())
    .mulDiv(RANGE.price(true, false), 10**PRICE.decimals());
```

Example: Trying to sell `1e-6=0.000001` OHM (9 decimals) for USDC (6 decimals) always leads to receiving `0`, regardless of the price, because `amountOut = amountIn_.mulDiv(10**reserve.decimals(), 10**ohm.decimals()) = 10^3 * 10^6 / 10^9 = 0`.

Recommendation: Consider only doing a single (full) division.

```
function getAmountOut(ERC20 tokenIn_, uint256 amountIn_)
    public
    view
    returns (uint256)
{
    if (tokenIn_ == ohm) {
        /// Calculate amount out
        uint256 amountOut = amountIn_
+           .mulDiv(
+             10**reserve.decimals() * RANGE.price(true, false),
+             10**ohm.decimals() * 10**PRICE.decimals()
+           )
-           .mulDiv(10**reserve.decimals(), 10**ohm.decimals())
-           .mulDiv(RANGE.price(true, false), 10**PRICE.decimals());
        ...
    }
    ... // apply similar code for the other case
}
```

The individual operands should not overflow for tokens with reasonable decimals & reasonable prices. (For 18 decimal tokens, the price can be `<= 1e58`.)

Olympus: This has been fixed, Operator.sol#L756

Spearbit: Acknowledged.

5.4.11 Wrong `Regen` initialization in `Operator.constructor`

Severity: *Low Risk*

Context: Operator.sol#L91

Description: The `Regen` regeneration struct is initialized with `observations.length = configParams[5]`. However, `configParams[5]` refers to `regenWait` which is a time duration in seconds.

> The impact of the wrong initialization is low because the structure is initialized a second time in the `initialize` call.

Recommendation: It should be `regenObserve = configParams[7]`.

```
Regen memory regen = Regen({
-   count: configParams[5],
+   count: 0,
    lastRegen: uint48(block.timestamp),
    nextObservation: uint32(0),
-   observations: new bool[](configParams[5])
+   observations: new bool[](configParams[7])
});
```

Consider using named parameters instead of indexing an opaque array of configuration parameters. Alternatively, consider destructuring the `configParams` array into named local variables at the beginning of the constructor and only use these named arguments.

Olympus: Fixed in Operator.sol#L115-L120.

Spearbit: Acknowledged.

5.4.12 Malicious policies may keep roles even after termination

Severity: *Low Risk*

Context: Kernel.sol#L217

Description: When terminating a policy the `Kernel` removes all of the policies requested roles. However, the requested roles are fetched from the policy itself. Malicious policies can return a different set of roles when being terminated than what they requested when being approved.

Recommendation: Consider storing the requested roles when a specific policy is approved. When terminating the policy again, iterate over granted roles and remove them.

Olympus: Acknowledged. We are changing the Kernel and will likely be reconfiguring the role system. Will be out of scope for this audit unfortunately.

Spearbit: Acknowledged.

5.5 Gas Optimization

5.5.1 Change `currentPrice` to optimized calculation

Severity: *Gas Optimization*

Context: PRICE.sol#L163-L167

Description: The `currentPrice` calculation uses `mulDiv` which is quite expensive and could be avoided cheaply.

Let's start with a simpler and more intuitive notation:

```
current price =
  (ohmDecimalPrice / reserveDecimalPrice) * 10^decimals
```

.. where the fraction should have 18 decimals of precision (decimals is a constant: 18)

But that cannot work in solidity since we don't have floats, so we can rewrite, naively: `10**(decimals + _reserveEthDecimals - _ohmEthDecimals) * ohmPrice / reservePrice`

What can go wrong with this version ? For that exponent to get negative, reserve price would need to use 0 decimals while ohm price uses more than 18. It seems like a far fetched edge case but we will handle it anyway. It is reasonable to assume that ohmPrice will not exceed 10**36, keeping it bound under 128bits. It also seems reasonable to assume that the exponent will not exceed 36 either (would require `_reserveEthDecimals > 18+_ohmEthDecimals`), but the edge case should be handled too (edge case 2). Multiplying those two <128bit values yields a result that fits in 256bits without overflow, and that result is already scaled up by `decimals`, keeping the desired precision when we finally divide by reservePrice.

Therefore, the expensive mulDiv call can be avoided, provided we handle the 2 edge cases. Moreover, the exponent is constant since all of its components are (should be) immutable. Its result can therefore be calculated once on deployment and stored as immutable too.

Recommendation: To reduce the gas fees consider following these recommendations.

Add this `scaleFactor` variable

```
+ uint256 internal immutable scaleFactor;
```

In`constructor()`, after all 3 components have been assigned, if this reverts with underflow or exp too large, consider changing `decimals` value.

```
+ uint256 exp = decimals + _reserveEthDecimals - _ohmEthDecimals; //  (negative exp edge case handled)
+ require(exp <= 38); // to keep scaleFactor under 128bits. (Edge case 2)

+ scaleFactor = 10**exp;
```

in `getCurrentPrice()` :

```
- uint256 currentPrice = ohmEthPrice.mulDiv(
-   10**(decimals + _reserveEthDecimals),
-   reserveEthPrice * 10**(_ohmEthDecimals)
- );
+ currentPrice = scaleFactor * ohmEthPrice / reserveEthPrice;
```

Olympus: This has now been applied. PRICE.sol#L93 PRICE.sol#L185

Spearbit: Acknowledged.

5.5.2 Cache OHM/Reserve decimals to `Operator`

Severity: *Gas Optimization*

Context: Operator.sol#L68-L72

Description: OHM and Reserve decimals can be cached for gas optimization since they won't change.

Recommendation: Consider adding the following lines.

```
/// Tokens
/// @notice     OHM token contract
ERC20 public immutable ohm;
+ /// @notice OHM decimals
+ uint256 public immutable ohmDecimals;

/// @notice     Reserve token contract
ERC20 public immutable reserve;
+ /// @notice Reserve decimals
+ uint256 public immutable reserveDecimals;
```

Olympus: Fixed, OlympusDAO/bophades2@7d03988 (#38)

Spearbit: Acknowledged.

5.5.3 `Price` module can use a ring buffer

Severity: *Gas Optimization*

Context: PRICE.sol#L137

Description: The `PRICE` module only ever needs the last `numObservations` data points to compute the moving average. However, it stores every single data point. Note that reusing existing non-zero storage slots costs a lot less gas than initializing a new slot each time.

Recommendation: Consider using a ring buffer implementation of length `numObservations` to save gas. The implementation could look similar to Operator's `observation` array. Moving to a ring buffer also requires changes to `changeMovingAverageDuration` as it takes advantage of the overstoring of data.

Olympus: Agree that this would save gas. We looked at doing something like this at one point, but a comment was made about preserving data for the moving average...

We discussed this again and decided to update to a "ring buffer" implementation to save gas. Observations emit an event and so the historical data can be tracked off chain. Updated here: PRICE.sol

Spearbit: Acknowledged.

5.5.4 Certain math operations can be unchecked

Severity: *Gas Optimization*

Context: TRSRY.sol#L124, TRSRY.sol#L162, PRICE.sol#L256

Description: The subtractions of `a - b` in the referenced lines can be wrapped in `unchecked {}` blocks because the condition that `a >= b` has been performed before.

Recommendation: Wrap the subtractions in `unchecked {}` blocks.

Olympus: Fixed. Implemented in TRSRY.sol#L164-L179 and changed the logic on PRICE.sol.

Spearbit: Acknowledged.

5.5.5 Use cached approval variable instead of re-reading from storage

Severity: *Gas Optimization*

Context: TRSRY.sol#L162

Description: The `withdrawApproval[msg.sender][token_]` has been read once and stored in the `approval` variable. It would be cheaper to re-use this variable like in `withdrawReserves()`.

Recommendation: Consider applying the following change.

```
- withdrawApproval[msg.sender][token_] -= amount_;
+ withdrawApproval[msg.sender][token_] = approval - amount_;
```

Olympus: Fixed. Implemented in TRSRY.sol#L164-L179.

Spearbit: Acknowledged.

5.5.6 Use postfix increment and cache `totalInstructions`

Severity: *Gas Optimization*

Context: INSTR.sol#L77, INSTR.sol#L80-L82, INSTR.sol#L120-L122

Description: It will be worth caching the value of `totalInstructions` after incrementing it since it is used multiple times inside the function.

Recommendation: Consider applying the following changes.

```
- totalInstructions++;
+ uint256 totalInstructionsCached = ++totalInstructions;

// initialize an empty list of instructions that will be filled
Instruction[] storage instructions = storedInstructions[
-    totalInstructions
+    totalInstructionsCached
];

...

- emit InstructionsStored(totalInstructions);
+ emit InstructionsStored(totalInstructionsCached);

- return totalInstructions;
+ return totalInstructionsCached;
```

Olympus: Fixed. Implemented in INSTR.sol#L70-L122.

Spearbit: Acknowledged.

5.5.7 else if (!for_) can be simplified to else case

Severity: *Gas Optimization*

Context: Governance.sol#L242

Description: Since `for_` is boolean, `else if (!for_)` can be simplified to `else`.

Recommendation:

```
- } else if (!for_) {
+ } else {
```

Olympus: Fixed. Implemented in Governance.sol#L243-L247.

Spearbit: Acknowledged.

5.5.8 Constant or immutable declarations

Severity: *Gas Optimization*

Context: OlympusERC20.sol#L55 PRICE.sol#L86 PRICE.sol#L35-L38

Description: State variables can be declared as constant or immutable. In both cases variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable it can still be assigned during constructor execution. Using these keywords also greatly reduces gas costs.

- OlympusERC20.sol#L55
- PRICE.sol#L86
- PRICE.sol#L35-L38

Recommendation: Declare these state variables as immutable or constant. If it is necessary to change state variable on constructor, it is recommended to use immutable. Check examples below:

```
- string UNAUTHORIZED = "UNAUTHORIZED";
+ string constant UNAUTHORIZED = "UNAUTHORIZED";
```

```
- uint8 public decimals;
+ uint8 public constant decimals = 18;
```

Olympus: We will not be modifying OlympusERC20.sol since it is a legacy contract that has already been deployed to the chain. We're keeping it in the repo as a flattened file to be able to reference as needed (like in MINTR.sol).Changes implemented in PRICE.sol#L63-L65 and PRICE.sol#L36-L37.

Spearbit: Acknowledged.

5.5.9 Pack variables in PRICE module

Severity: *Gas Optimization*

Context: PRICE.sol#L48-L64

Description: Moving `_ohmEthDecimals` and `_reserveEthDecimals` declartions before `decimals` will make the variables from `observationFrequency` to `initialized` be in just one storage slot.

Recommendation: Consider aplying the following changes.

```
...
AggregatorV2V3Interface internal _ohmEthPriceFeed;
AggregatorV2V3Interface internal _reserveEthPriceFeed;
- uint8 internal _ohmEthDecimals;
- uint8 internal _reserveEthDecimals;
...
/// @notice Unix timestamp of last observation (in seconds).
uint48 public lastObservationTime;

/// @notice Number of decimals in the price values provided by the contract.
uint8 public decimals;
+ uint8 internal _ohmEthDecimals;
+ uint8 internal _reserveEthDecimals;
...
```

Olympus: Fixed. Implemented in PRICE.sol#L63-L65.

Spearbit: Acknowledged.

5.5.10 Checking `depositInterval_` before `duration_` reduces number of checks needed

Severity: *Gas Optimization*

Context: Operator.sol#L473-L477

Description: The bounds for `depositInterval_` and `duration_` are: `1 hours <= depositInterval_ <= duration_ <= 7 days`. Thus, only 3 checks need to be performed. By shifting the `depositInterval_` checks above `duration_`, one check can be removed.

Recommendation: Consider implementing the following changes.

```
+ if (depositInterval_ < uint32(1 hours) || depositInterval_ > duration_)
+   revert Operator_InvalidParams();
- if (duration_ > uint256(7 days) || duration_ < uint256(1 hours))
+ if (duration_ > uint256(7 days))
revert Operator_InvalidParams(); // TODO validate these bounds for duration
if (debtBuffer_ < uint32(10_000)) revert Operator_InvalidParams();
- if (depositInterval_ < uint32(1 hours) || depositInterval_ > duration_)
-   revert Operator_InvalidParams();
```

Olympus: That would work if those were the correct bounds. Reviewing this made me realize, it should be 1 days <= duration <= 7 days. Therefore, it is an issue, but different than described. Fixed in Operator.sol#L537-L541.

Spearbit: Acknowledged.

5.5.11 Boolean comparisons

Severity: *Gas Optimization*

Context: Kernel.sol#L195 Kernel.sol#L224 TRSRY.sol#L135 Governance.sol#L204 Governance.sol#L308 Kernel.sol#L48 Kernel.sol#L210

Description: Comparing a boolean to a constant (`true` or `false`) is a bit more expensive than directly checking the boolean value. Recommend doing the latter.

Recommendation: Consider implementing the following changes.

```
- if(x == true)
+ if(x)

- if(y == false)
+ if(!y)
```

Olympus: Fixed in various files and commits.

Spearbit: Acknowledged.

5.5.12 For Loop Optimization

Severity: *Gas Optimization*

Context: Kernel.sol#L221 Kernel.sol#L236 INSTR.sol#L90 INSTR.sol#L138 PRICE.sol#L212 PRICE.sol#L258 BondCallback.sol#L133 Governance.sol#L284

Description: The for-loop can be optimised in 4 ways:

- Removing initialization of loop counter if value is 0 by default.
- Caching array length outside the loop.
- Prefix increment (`++i`) instead of postfix increment (`i++`).
- Unchecked increment.

Specifically, for PRICE.sol#L258, offsetting the counter initially in the initialization saves gas becuase it avoids applying the offset in each iteration.

```
- for (uint256 i; i < newObservations; ++i) {
+ for (uint256 i = startIdx; i < obsLength; ) {
-   newMovingAverage += observations[startIdx + i];
+   newMovingAverage += observations[i];
+   unchecked {
+     ++i
+   }
```

Recommendation: Optimize for-loops.

Olympus: Fixed, for loops have been optimised.

Spearbit: Acknowledged.

5.6 Informational

5.6.1 Consider zeroing `lastObservationTime` in `PRICE`

Severity: *Informational*

Context: PRICE.sol#L250-L253

Description: For accuracy, whenever `initialized` is set to `false`, consider zeroing `lastObservationTime` as well.

Recommendation: Consider adding the following line.

```
observations = new uint256[](newObservations);
/// Set initialized to false
initialized = false;
+ lastObservationTime = 0;
```

Olympus: Fixed. Implemented in PRICE.sol#L265 and PRICE.sol#L302

Spearbit: Acknowledged.

5.6.2 `BondCallback` doesn't have function to remove whitelisted markets

Severity: *Informational*

Context: BondCallback.sol#L67-L76

Description: The `whitelist()` function only enables whitelisting with `teller_` and `id_` parameters. But there aren't any additional features in the contract to set the `approvedMarkets[teller_][id_]` variable to `false`.

Recommendation: One approach is using the following code.

```
- function whitelist(address teller_, uint256 id_)
+ function whitelist(address teller_, uint256 id_, bool enable_)
        external
        override
        requiresAuth
    {
-        approvedMarkets[teller_][id_] = true;
+        approvedMarkets[teller_][id_] = enable_;
    }
```

Olympus: Since a market has a finite duration and uses a callback address for payouts (if provided) it does not make sense to un-whitelist a market. It would no longer be able to function anyways. Therefore, the owner could just close the market on the BondAuctioneer.

Spearbit: Acknowledged.

5.6.3 Treasury rejects ETH transfers

Severity: *Informational*

Context: TRSRY.sol#L93-L95

Description: ETH transferred to the contract accidentally or intentionally will be rejected on the contract since the `receive()` function on the contract has been commented out.

Recommendation: If ETH is supposed to be accepted as a treasury asset, either:

1. Add an empty `receive` function and functions to withdraw ETH again. Currently, only ERC20 tokens can be withdrawn.
2. Wrap any received ETH directly into WETH:

```
receive() external payable {
    WETH(WETH_ADDRESS).deposit{value: msg.value}();
}
```

Olympus: We removed the WETH dependencies and the payable function completely. It should just reject any ETH transfers now. Fixed, implemented in TRSRY.sol.

Spearbit: Acknowledged.

5.6.4 Protocol does not support fee-on-transfer tokens

Severity: *Informational*

Context: TRSRY.sol#L184

Description: Some ERC20 tokens make modifications to their `transfer` or `balanceOf` functions. One of these tokens include deflationary tokens that charge a fee on every `transfer()` and `transferFrom()`. The protocol does not handle fee-on-transfer tokens well. When repaying debt to the treasury the pre-fee amount is deducted but the received amount might be lower.

Recommendation: If support for fee-on-transfer tokens is desired, consider using pre-and-post balances to compute the actual received amount.

```
function repayLoan(ERC20 token_, uint256 amount_)
    external
    nonReentrant // @audit must also be non-reentrant now to avoid ERC777 token exploits
    onlyRole(BANKER)
{
    // Deposit from caller
    uint256 receivedAmount = token_.balanceOf(address(this));
    token_.safeTransferFrom(msg.sender, address(this), amount_);
    receivedAmount = token_.balanceOf(address(this)) - receivedAmount;

    // Subtract debt to caller
    reserveDebt[token_][msg.sender] -= receivedAmount;
    totalDebt[token_] -= receivedAmount;

    emit DebtRepaid(token_, msg.sender, receivedAmount);
}
```

Olympus: Fixed. Implemented in TRSRY.sol#L129-L145.

Spearbit: Acknowledged.

5.6.5 Safety check `payoutToken == OHM`

Severity: *Informational*

Context: BondCallback.sol#L111-L114

Description: It's unclear if `payoutToken` could ever be anything other than OHM in the `else` case, consider checking `payoutToken == OHM` to be safe.

Recommendation: Add the following line as follows:

```
else {
    // Else (selling ohm), mint OHM to sender
+   if (payoutToken != OHM) { revert }
    ...
}
```

Olympus: Fixed. Implemented in BondCallback.sol#L122 and BondCallback.sol#L128.

Spearbit: Acknowledged.

5.6.6 Not fixed contract pragma versions

Severity: *Informational*

Context: All contracts

Description: Contracts don't have a fixed pragma version and could cause unexpected errors.

Recommendation: Change to a fixed pragma version.

```
- pragma solidity ^0.8.13;
+ pragma solidity 0.8.13;
```

Olympus: Fixed, pragma versions have been locked for contracts.

Spearbit: Acknowledged.

5.6.7 Sanity check for `startObservations` in PRICE module

Severity: *Informational*

Context: PRICE.sol#L211-L215

Description: Add sanity check to `startObservations` array to not lead to an invalid `_movingAverage`

Recommendation: Consider applying the following change:

```
uint256 total;
for (uint48 i; i < numObs; ++i) {
+  if (startObservations_[i] == 0) revert Price_InvalidParams();
    total += startObservations_[i];
    observations[i] = startObservations_[i];
}
```

Olympus: Fixed PRICE.sol#L234.

Spearbit: Acknowledged.

5.6.8 Unused imports and redundancies

Severity: *Informational*

Context: Operator.sol#L16 Operator.sol#L45 AUTHR.sol#L6 TRSRY.sol#L10-L12 TRSRY.sol#L15 TRSRY.sol#L20 TRSRY.sol#L248-258 Heart.sol#L28 INSTR.sol#L37 VoterRegistration.sol#L43-L47 TransferHelper.sol#L61-L65

Description: Some imports, events, errors and variables are unused. It is recommended to clear out unused objects from the code to improve readability. Additionally, it is also a best practice to remove redundancies from contracts to reduce gas consumption during deployment and execution time.

- Operator.sol#L16 - Unused import
- Operator.sol#L45 - Unused event
- AUTHR.sol#L6 - Unused import
- TRSRY.sol#L10-L12 - Unused interface
- TRSRY.sol#L15 - Unused error
- TRSRY.sol#L20 - Unused variable
- TRSRY.sol#L248-258 - Redundant function
- Heart.sol#L28 - Unused error
- INSTR.sol#L37 - Unused event
- VoterRegistration.sol#L43-L47 - Unused event and error
- TransferHelper.sol#L61-L65 - Unused function

Recommendation: Remove unused objects and redundancies.

Olympus: Latest commit in develop branch has all of the remaining unused items removed. TRSRY, removed all unused events and errors , and moved the revokePolicyApproval to the TreasuryCustodian.sol policy - TRSRY.sol

INSTR, removed all unused errors and events - INSTR.sol

VoterRegistration, removed all events and errors - VoterRegistration.sol

TransferHelper, removed safeTransferETH function - TransferHelper.sol

Operator, emit Swap event when swap function is called - Swap event

Spearbit: Acknowledged.

5.6.9 Comment clarifications and typos

Severity: *Informational*

Context: IOperator.sol#L15 PRICE.sol#L75 Operator.sol#L248 Operator.sol#L559 Operator.sol#L688 Governance.sol#L208 Governance.sol#L227 VoterRegistration.sol#L3 INSTR.sol#L41-L47 TRSRY.sol#L189-L200 Governance.sol#L63-L72

Description: Collection of incorrect comments or improved for greater clarity, and spelling, grammar and typo errors

Recommendation: Consider applying the folling changes in:

`IOperator.sol`

```
- at anyone time
+ at any one time
```

`PRICE.sol`

```
- divislble
+ divisible
```

Operator.sol

```
- /// Revert if lower wall is inactive
+ /// Revert if higher wall is inactive


- sicne
+ since



- /// Revert if amount in exceeds capacity
+ /// Revert if amount out exceeds capacity
if (amountOut > RANGE.capacity(true))
```

Governance.sol

```
- ensure the current active proposal has had at least a week of voting
+ ensure the current active proposal has had at least a week of voting and execution time


- // // ensure an active proposal exists
+ // ensure an active proposal exists
```

VoterRegistration.sol

```
- // The Proposal Policy submits & activates instructions in a INSTR module
+ // TODO: Add description for VoterRegistration policy
```

INSTR.sol

```
enum Actions {
-  ChangeExecutive,
+ InstallModule,
+ UpgradeModule,
ApprovePolicy,
TerminatePolicy,
-  InstallSystem,
-  UpgradeSystem
+ ChangeExecutor
}
```

TRSRY.sol

```
- // TODO for repaying debt in different tokens. Specifically for changing reserve assets
- /*
- function repayDebtEquivalent(
-     ERC20 origToken_,
-     ERC20 repayToken_,
-     uint256 debtAmount_
- ) external onlyPermittedPolicies {
-     // TODO reduce debt amount of original token
-     reserveDebt[origToken_][msg.sender] -= debtAmount_;
-     totalDebt[origToken_] -= debtAmount_;
- }
- */
```

Governance.sol

```
function requestRoles() external view override onlyKernel returns (Kernel.Role[] memory roles) {
+ //should also be the kernel.executor
   roles = new Kernel.Role[](1);
   ...
}
```

Olympus: These have all been addressed by the latest commit 72a3a4 and 907d82

Spearbit: Acknowledged.

5.6.10 Redundant `onlyKernel()` modifier in policy's `configureReads()`

Severity: *Informational*

Context: BondCallback.sol#L49 Operator.sol#L113

Description: The `BondCallback` and `Operator` policies have their `configureReads()` function restricted by the `onlyKernel` modifier while the rest are not. This isn't strictly necessary as `setAuthority()` will be called (at least for the policies mentioned), which is permissioned: either `msg.sender` has to be the owner, or, if the authority has already been set, requires `authority.canCall(msg.sender, address(this), msg.sig)`.

In general, as the function is fetching values that are set in the kernel, making the function permissionless shouldn't pose much danger.

Recommendation: Remove the `onlyKernel` modifier.

Olympus: Fixed.

Spearbit: Fixed in BondCallback.sol#L52 and Operator.sol#L138

5.6.11 Upgrading modules can be dangerous

Severity: *Informational*

Context: Kernel.sol#L191

Description: Governance actions can upgrade module contracts and all policies are reconfigured to use the new module. Replacing a policy's module is like replacing their storage data and could break a lot of the policy's functionality.

Recommendation: Care must be taken when upgrading modules. We recommend simulating the effects of any governance actions on the system.

Olympus: We are changing the Kernel which will be out of scope for this particular audit.

Spearbit: Acknowledged.

5.6.12 Misleading `Kernel_ModuleAlreadyExists` error

Severity: *Informational*

Context: Kernel.sol#L185

Description: If `oldModule == address(0)`, the code throws a `Kernel_ModuleAlreadyExists` error but the module does not yet exist.

Recommendation: It should say that no module was found to upgrade in this case.

Olympus: Fixed in Kernel.sol#L182

Spearbit: Acknowledged.

5.6.13 Redundant `onlyKernel()` modifier in policy's `requestRoles()`

Severity: *Informational*

Context: BondCallback.sol#L59 Governance.sol#L67 Heart.sol#L87 Operator.sol#L126 PriceConfig.sol#L32 VoterRegistration.sol#L32

Description: The policies' `requestRoles()` function is restricted by the `onlyKernel` modifier and because it is a view function that merely returns the roles that the policy requires from the kernel, the data is not particularly sensitive and can therefore be unrestricted.

Recommendation: Remove the `onlyKernel` modifier.

Olympus: Fixed, onlyKernel modifier has been removed.

Spearbit: Acknowledged.

5.6.14 Add check and revert reason if no votes > yes votes in propsal execution

Severity: *Informational*

Context: Governance.sol#L266-L267

Description: If no votes > yes votes the attempted proposal execution will revert due to subtraction overflow.

Recommendation: Consider adding a check and revert reason for clarity.

Olympus: As discussed in the PR, we opted to not add the check because we only need the function to execute in the success case, and the extra check only punishes users that successfully execute proposals. Adding the check does not modify the behavior of the function under any circumstances as far as we are aware.

Spearbit: Acknowledged, check has not been implemented.